**DATABASE ARCHITECTURE**

Every database, and indeed every DBMS, must adhere to the principles of some data model. However, the term data model is somewhat ambiguous. In the database system literature the term is used in a number of different senses, two of which are the most important: that of architecture for data; that of an integrated set of data requirements.

**Data Model as Architecture**

In this sense of the term, data model is used to refer to a set of general principles for handling data. Here, people talk of the relational data model, the hierarchical data model or Object oriented data model.

The set of principles that defines a data model may be divided into three major parts:

1. Data definition - A set of principles concerned with how data is structured.

2. Data manipulation - A set of principles concerned with how data is operated on.

3. Data integrity - A set of principles concerned with determining which states are valid for a database.

Data definition involves defining an organization for data: a set of templates into which data will be fitted.

Data manipulation concerns the process of how the data is accessed and how it is changed in the database.

Data integrity is very much linked with the idea of data manipulation in the sense that integrity concerns the idea of what are valid changes and invalid changes to data.

Any database and DBMS must adhere to the tenets of some data model. Hence, in the relational data model, data definition involves the concept of a relation, data manipulation involves a series of relational operators, and data integrity amounts to two rules - entity and referential integrity. Note that by the data integrity part of a data model we are describing only those rules those are inherently part of the data model. A great deal of other integrity constraints or rules will have to be specified by additional means, i.e. using mechanisms not inherently part of the data model.

**Data Model as Blueprint/plan/outline**

In this sense, the term data model is used to refer to an integrated, but implementation independent, set of data requirements for some application. Here, analysts might speak of the order-processing data model, the accounts-receivable data model, or the student admissions data model. A data model in this sense is an important component part of any information systems specification. To avoid confusion, whenever it is not explicit from the context we shall prefix the first type of data model with the word architectural and the second type of data model with the word applications.

**Topology of Architectural Data Models**

We may make a distinction between three generations of architectural data model

**1. Primitive Data Models.**

In this approach objects are represented by record-structures grouped in file-structures.

The main operations available are read and write operations over records.

**2. Classic Data Model.**

These are the hierarchical, network and relational data models. The hierarchical data model is an extension of the primitive data model discussed above. The network is an extension of the hierarchical approach. The relational data model is a fundamental departure from the hierarchical and network approaches.

**3. Semantic Data Models**

The main problem with classic data models like the relational data model is that they maintain a fundamental record-orientation. In other words, the meaning of the information in the database - its semantics – is not readily apparent from the database itself. Semantic information must be consciously applied by the user of databases using the classic approach. For this reason, a number of so-called semantic data models have been proposed.

Semantic data models (SDMs) attempt to provide a more expressive means of representing the meaning of information that is available in the classic models. In many senses the object-oriented data model can be regarded as a semantic data model.

**Types of Database Models**

**Hierarchical Model**

The hierarchical data model organizes data in a tree structure. There is a hierarchy of parent and child data segments. This structure implies that a record can have repeating information, generally in the child data segments. Data in a series of records, which have a set of field values attached to it. It collects all the instances of a specific record together as a record type. These record types are the equivalent of tables in the relational model, and with the individual records being the equivalent of rows. To create links between these record types, the hierarchical model uses Parent Child Relationships. These are a 1:N mapping between record types. This is done by using trees, like set theory used in the relational model, "borrowed" from maths. For example, an organization might store information about an employee, such as name, employee number, department, salary. The organization might also store information about an employee's children, such as name and date of birth. The employee and children data forms a hierarchy, where the employee data represents the parent segment and the children data represents the child segment. If an employee has three children, then there would be three child segments associated with one employee segment. In a hierarchical database the parent-child relationship is one to many. This restricts a child segment to having only one parent segment. Hierarchical DBMSs were popular from the late 1960s, with the introduction of IBM's Information Management System (IMS) DBMS, through the 1970s.

**Network Model**

The popularity of the network data model coincided with the popularity of the hierarchical data model. Some data were more naturally modeled with more than one parent per child. So, the network model permitted the modeling of many-to-many relationships in data. In 1971, the Conference on Data Systems Languages (CODASYL) formally defined the network model. The basic data modeling construct in the network model is the set construct. A set consists of an owner record type, a set name, and a member record type. A member record type can have that role in more than one set, hence the multi parent concept is supported. An owner record type can also be a member or owner in another set. The data model is a simple network, and link and intersection record types (called junction records by IDMS) may exist, as well as sets between them. Thus, the complete network of relationships is represented by several pairwise sets; in each set some (one) record type is owner (at the tail of the network arrow) and one or more record types are members (at the head of the relationship arrow). Usually, a set defines a 1:M relationship, although 1:1 is permitted. The CODASYL network model is based on mathematical set theory.

**Relational Model**

(RDBMS - relational database management system) A database based on the relational model developed by E.F. Codd. A relational database allows the definition of data structures, storage and retrieval operations and integrity constraints. In such a database the data and relations between them are organized in tables. A table is a collection of records and each record in a table contains the same fields.

Certain fields may be designated as keys, which mean that searches for specific values of that field will use indexing to speed them up. Where fields in two different tables take values from the same set, a join operation can be performed to select related records in the two tables by matching values in those fields. Often, but not always, the fields will have the same name in both tables. For example, an "orders" table might contain (customer-ID, product-code) pairs and a "products" table might contain (product-code, price) pairs so to calculate a given customer's bill you would sum the prices of all products ordered by that customer by joining on the product-code fields of the two tables. This can be extended to joining multiple tables on multiple fields. Because these relationships are only specified at retreival time, relational databases are classed as dynamic database management system. The RELATIONAL database model is based on the Relational Algebra.

**Object/Relational Model**

Object/relational database management systems (ORDBMSs) add new object storage capabilities to the relational systems at the core of modern information systems. These new facilities integrate management of traditional fielded data, complex objects such as time-series and geospatial data and diverse binary media such as audio, video, images, and applets. By encapsulating methods with data structures, an ORDBMS server can execute complex analytical and data manipulation operations to search and transform multimedia and other complex objects.

As an evolutionary technology, the object/relational (OR) approach has inherited the robust transaction- and performance-management features of its relational ancestor and the flexibility of its object-oriented cousin. Database designers can work with familiar tabular structures and data definition languages (DDLs) while assimilating new object-management possibilities. Query and procedural languages and call interfaces in ORDBMSs are familiar: SQL3, vendor procedural languages, and ODBC, JDBC, and proprietary call interfaces are all extensions of RDBMS languages and interfaces. And the leading vendors are, of course, quite well known: IBM, Inform ix, and Oracle.

**Object-Oriented Model**

Object DBMSs add database functionality to object programming languages. They bring much more than persistent storage of programming language objects. Object DBMSs extend the semantics of the C++, Smalltalk and Java object programming languages to provide full-featured database programming capability, while retaining native language compatibility. A major benefit of this approach is the unification of the application and database development into a seamless data model and language environment. As a result, applications require less code, use more natural data modeling, and code bases are easier to maintain. Object developers can write complete database applications with a modest amount of additional effort.

According to Rao (1994), "The object-oriented database (OODB) paradigm is the combination of object-oriented programming language (OOPL) systems and persistent systems. The power of the OODB comes from the seamless treatment of both persistent data, as found in databases, and transient data, as found in executing programs."

In contrast to a relational DBMS where a complex data structure must be flattened out to fit into tables or joined together from those tables to form the in-memory structure, object DBMSs have no performance overhead to store or retrieve a web or hierarchy of interrelated objects. This one-to-one mapping of object programming language objects to database objects has two benefits over other storage approaches: it provides higher performance management of objects, and it enables better management of the complex interrelationships between objects. This makes object DBMSs better suited to support applications such as financial portfolio risk analysis systems, telecommunications service applications, World Wide Web document structures, design and manufacturing systems, and hospital patient record systems, which have complex relationships between data.

**Semi structured Model**

In semi structured data model, the information that is normally associated with a schema is contained within the data, which is sometimes called ``self-describing''. In such database there is no clear separation between the data and the schema, and the degree to which it is structured depends on the application. In some forms of semi structured data there is no separate schema, in others it exists but only places loose constraints on the data. Semi-structured data is naturally modelled in terms of graphs which contain labels which give semantics to its underlying structure. Such databases subsume the modelling power of recent extensions of flat relational databases, to nested databases which allow the nesting (or encapsulation) of entities, and to object databases which, in addition, allow cyclic references between objects.

Semi structured data has recently emerged as an important topic of study for a variety of reasons. First, there are data sources such as the Web, which we would like to treat as databases but which cannot be constrained by a schema. Second, it may be desirable to have an extremely flexible format for data exchange between disparate databases. Third, even when dealing with structured data, it may be helpful to view it as semi structured for the purposes of browsing.

**Associative Model**

The associative model divides the real-world things about which data is to be recorded into two sorts:

Entities are things that have discrete, independent existence. An entity’s existence does not depend on any other thing. Associations are things whose existence depends on one or more other things, such that if any of those things ceases to exist, then the thing itself ceases to exist or becomes meaningless.

An associative database comprises two data structures:

1. A set of items, each of which has a unique identifier, a name and a type.

2. A set of links, each of which has a unique identifier, together with the unique identifiers of three other things, which represent the source verb and target of a fact that is recorded about the source in the database. Each of the three things identified by the source, verb and target may be either a link or an item.

**Entity-Attribute-Value (EAV) data model**

The best way to understand the rationale of EAV design is to understand row modeling (of which EAV is a generalized form). Consider a supermarket database that must manage thousands of products and brands, many of which have a transitory existence. Here, it is intuitively obvious that product names should not be hard-coded as names of columns in tables. Instead, one stores product descriptions in a Products table: purchases/sales of individual items are recorded in other tables as separate rows with a product ID referencing this table. Conceptually an EAV design involves a single table with three columns, an entity (such as an olfactory receptor ID), an attribute (such as species, which is actually a pointer into the metadata table) and a value for the attribute (e.g., rat). In EAV design, one row stores a single fact. In a conventional table that has one column per attribute, by contrast, one row stores a set of facts. EAV design is appropriate when the number of parameters that potentially apply to an entity is vastly more than those that actually apply to an individual entity.

**Context Model**

The context data model combines features of all the above models. It can be considered as a collection of object-oriented, network and semi structured models or as some kind of object database. In other words this is a flexible model, you can use any type of database structure depending on task. Such data model has been implemented in DBMS Context.

The fundamental unit of information storage of Context is a CLASS. Class contains METHODS and describes OBJECT. The Object contains FIELDS and PROPERTY. The field may be composite, in this case the field contains Sub Fields etc. The property is a set of fields that belongs to particular Object. (similar to AVL database). In other words, fields are permanent part of Object but Property is its variable part.

The header of Class contains the definition of the internal structure of the Object, which includes the description of each field, such as their type, length, attributes and name. Context data model has a set of predefined types as well as user defined types. The predefined types include not only character strings, texts and digits but also pointers (references) and aggregate types (structures).

**Types of Fields**

A context model comprises three main data types: REGULAR, VIRTUAL and REFERENCE. A regular (local) field can be ATOMIC or COMPOSITE. The atomic field has no inner structure. In contrast, a composite field may have a complex structure, and its type is described in the header of Class. The composite fields are divided into STATIC and DYNAMIC. The type of a static composite field is stored in the header and is permanent. Description of the type of a dynamic composite field is stored within the Object and can vary from Object to Object.

Like a NETWORK database, apart from the fields containing the information directly, context database has fields storing a place where this information can be found, i.e. POINTER (link, reference) which can point to an Object in this or another Class. Because main addressed unit of context database is an Object, the pointer is made to Object instead of a field of this Object. The pointers are divided on STATIC and DYNAMIC. All pointers that belong to a particular static pointer type point to the same Class (albeit, possibly, to different Object). In this case, the Class name is an integral part of the pointer type. A dynamic pointer type describes pointers that may refer to different Classes. The Class, which may be linked through a pointer, can reside on the same or any other computer on the local area network. There is no hierarchy between Classes and the pointer can link to any Class, including its own.

In contrast to pure object-oriented databases, context databases is not so coupled to the programming language and doesn't support methods directly. Instead, method invocation is partially supported through the concept of VIRTUAL fields.

A VIRTUAL field is like a regular field: it can be read or written into. However, this field is not physically stored in the database, and in it does not have a type described in the scheme. A read operation on a virtual field is intercepted by the DBMS, which invokes a method associated with the field and the result produced by that method is returned. If no method is defined for the virtual field, the field will be blank. The METHODS is a subroutine written in C++ by an application programmer. Similarly, a write operation on a virtual field invokes an appropriate method, which can changes the value of the field. The current value of virtual fields is maintained by a run-time process; it is not preserved between sessions. In object-oriented terms, virtual fields represent just two public methods: reading and writing. Experience shows, however, that this is often enough in practical applications. From the DBMS point of view, virtual fields provide transparent interface to such methods via an application written by application programer.

A context database that does not have composite or pointer fields and property is essentially RELATIONAL. With static composite and pointer fields, context database become OBJECT-ORIENTED. If the context database has only Property in this case it is an ENTITY-ATTRIBUTE-VALUE database. With dynamic composite fields, a context database becomes what is now known as a SEMISTRUCTURED database. If the database has all available types... in this case it is Context database.

**DATABASE SCHEMA AND INSTANCES**

The database schema of a database is set of definitions, which describe the structure of a given database. The schema of a database is also referred to as its intension. The activity of developing a schema for a database system is referred to as database design.

Below, for instance, we informally define the schema relevant to a university database:

**Schema: university**

**Classes:**

Modules - courses run by the institution in an academic semester

Students - people taking modules at the institution

**Relationships:**

Students take Modules

**Attributes:**

Modules have names

Students have names

In the schema we have identified classes of things such as modules, relationships between classes such as students take modules, and properties of classes such as modules have names.

# DISTRIBUTED DATABASE

A distributed database is a [database](http://en.wikipedia.org/wiki/Database) that is under the control of a central [database management system](http://en.wikipedia.org/wiki/Database_management_system) (DBMS) in which [storage devices](http://en.wikipedia.org/wiki/Computer_storage) are not all attached to a common [CPU](http://en.wikipedia.org/wiki/Central_processing_unit). It may be stored in multiple computers located in the same physical location, or may be dispersed over a [network](http://en.wikipedia.org/wiki/Computer_network) of interconnected computers.

Collections of data (e.g. in a database) can be distributed across multiple physical locations. A distributed database is distributed into separate partitions/fragments. Each partition/fragment of a distributed database may be replicated (i.e. [redundant](http://en.wikipedia.org/wiki/Redundancy_%28databases%29) fail-overs, [RAID](http://en.wikipedia.org/wiki/Redundant_array_of_independent_disks) like).

Besides distributed database replication and fragmentation, there are many other distributed database design technologies. For example, local autonomy, synchronous and asynchronous distributed database technologies. These technologies' implementation can and does depend on the needs of the business and the sensitivity/confidentiality of the data to be stored in the database, and hence the price the business is willing to spend on ensuring data security, consistency and integrity.

A database server is the software managing a database, and a client is an application that requests information from a server. Each computer in a system is a node. A node in a distributed database system acts as a client, a server, or both, depending on the situation like.

Horizontal fragments: subsets of tuples (rows) from a relation (table).

Vertical fragments: subsets of attributes (columns) from a relation (table).

Mixed fragment: a fragment which is both horizontally and vertically fragmented, or a logical collection of objects in an [ODBMS](http://en.wikipedia.org/wiki/ODBMS).

Homogeneous distributed database: uses one DBMS (eg: [Objectivity/DB](http://en.wikipedia.org/wiki/Objectivity/DB) or [Oracle](http://en.wikipedia.org/wiki/Oracle_database)).

Heterogeneous distributed database: uses multiple DBMS's (eg: Oracle and [MS-SQL](http://en.wikipedia.org/wiki/Microsoft_SQL_Server) and [PostgreSQL](http://en.wikipedia.org/wiki/PostgreSQL)).

Users access the distributed database through:

Local applications: applications which do not require data from other sites.

Global applications: applications which do require data from other sites.

## Advantages of distributed databases

* Reflects organizational structure — database fragments are located in the departments they relate to.
* Local autonomy — a department can control the data about them (as they are the ones familiar with it.)
* Improved availability — a fault in one database system will only affect one fragment, instead of the entire database.
* Improved performance — data is located near the site of greatest demand, and the database systems themselves are parallelized, allowing load on the databases to be balanced among servers. (A high load on one module of the database won't affect other modules of the database in a distributed database.)
* Economics — it costs less to create a network of smaller computers with the power of a single large computer.
* Modularity — systems can be modified, added and removed from the distributed database without affecting other modules (systems).

## 

## Disadvantages of distributed databases

* Complexity — extra work must be done by the [DBAs](http://en.wikipedia.org/wiki/Database_administrator) to ensure that the distributed nature of the system is transparent. Extra work must also be done to maintain multiple disparate systems, instead of one big one. Extra database design work must also be done to account for the disconnected nature of the database — for example, joins become prohibitively expensive when performed across multiple systems.
* Economics — increased complexity and a more extensive infrastructure means extra labour costs.
* Security — remote database fragments must be secured, and they are not centralized so the remote sites must be secured as well. The infrastructure must also be secured (e.g., by encrypting the network links between remote sites).
* Difficult to maintain integrity — in a distributed database, enforcing integrity over a network may require too much of the network's resources to be feasible.
* Inexperience — distributed databases are difficult to work with, and as a young field there is not much readily available experience on proper practice.
* Lack of standards – there are no tools or methodologies yet to help users convert a centralized DBMS into a distributed DBMS.
* Database design more complex – besides of the normal difficulties, the design of a distributed database has to consider fragmentation of data, allocation of fragments to specific sites and data replication.